Algoritmo de Subdivisão de Superfícies Deformáveis para Execução em GPU

Fernando Pedro Birra Manuel Próspero dos Santos

Departamento de Informática
Faculdade de Ciências e Tecnologia da
Universidade Nova de Lisboa
Quinta da Torre, 2825-114 Caparica
{fpb,ps}@di.fct.unl.pt

Sumário

A subdivisão recursiva de superfícies é uma técnica que permite obter uma superfície final suave, com elevado nível de detalhe, partindo de uma superfície inicial, de controlo, muito pouco detalhada. Se é verdade que, por um lado, esta técnica se torna muito apetecível efectuando simulações demoradas apenas numa malha de controlo, por outro, se essa mesma malha tiver que ser subdividida pelo CPU antes de ser passada ao GPU, para visualização, então perde-se grande parte ou até mesmo a totalidade das vantagens na sua utilização. Este artigo propõe um algoritmo de subdivisão de malhas regulares formadas por triângulos, agrupados dois a dois em quadriláteros. O algoritmo é executado na sua totalidade no GPU, aliviando o CPU para as tarefas de simulação. O exemplo que serve de ilustração a este algoritmo é o da simulação de superfícies deformáveis (p.ex. tecidos) com variação dinâmica do nível de detalhe. Trata-se de um domínio de aplicação onde todo o tempo de CPU se revela necessário para a avaliação do modelo mecânico subjacente à dinâmica da superfície deformável.

Palavras-chave

Computação no GPU, subdivisão, simulação, superfícies deformáveis, níveis de detalhe.

1. INTRODUÇÃO

A subdivisão de superfícies é reconhecida como sendo a técnica que preenche o vazio entre malhas de triângulos simples e os sofisticadas representações com NURBS, sendo usadas sempre que se pretende aliar a eficiência das malhas de triângulos com a qualidade superior das superfícies de NURBS.

Exemplos de esquemas de subdivisão abundam na literatura, tendo alguns deles sido exaustivamente estudados e analisados, tais como a subdivisão de Loop [Loop87], Doo Sabin [Doo78] ou Catmull-Clark [Catmull78]. Ao subdividir-se uma superfície, de forma recursiva, vão-se obtendo superfícies cada vez mais refinadas e que, caso o método de subdivisão seja bom, se vão aproximando de uma superfície limite suave.

Não menosprezando outras aplicações tão ou mais válidas, a utilização da subdivisão recursiva de superfícies é particularmente tentadora no caso da simulação de tecidos ou outras superfícies deformáveis.

A simulação absorve quase todo o tempo de CPU disponível e passa, em primeiro lugar, pela avaliação de um modelo matemático complexo que descreve o comportamento mecânico do tecido, traduzindo-se no cálculo de forças aplicadas em diversos pontos de amostragem. Normalmente aproximam-se os tecidos com malhas de polígonos formando os seus vértices um sistema de partículas. Posteriormente, obtém-se novas posições para os vértices da malha resolvendo um sistema de equações diferenciais.

A complexidade temporal dos algoritmos muito raramente é linear no número de partículas. Por estas razões é natural que se proceda à simulação de uma malha menos refinada.

Em [Bridson02] foi usado um passo de pós-processamento para suavizar as malhas usadas na simulação de tecidos. O algoritmo de subdivisão escolhido foi a subdivisão de Loop modificada [Loop01]. De modo a lidar com o problema das intersecções que se podem formar durante cada passo de subdivisão, as posições das partículas vão sendo ajustadas, caso necessário, de modo a garantir que a malha final de cada passo esteja isenta de intersecções com objectos sólidos. A malha inicial que é fornecida ao algoritmo de subdivisão é garantidamente uma malha isenta de intersecções ou seja, de colisões.

No entanto, a utilização de subdivisão de superfícies não garante, por si só, que a malha final possa capturar os detalhes que se obteriam usando uma malha mais refinada na simulação. É fundamental a utilização de malhas com suficiente nível de detalhe durante a simulação em determinadas zonas do tecido. Neste sentido é vantajoso usar-se simultaneamente um esquema de variação dinâmica do nível de detalhe, tal como em [Birra04].

Com o advento de processadores gráficos (GPUs) programáveis e com alto desempenho, tais como os disponíveis na quase generalidade das placas gráficas comercializadas hoje em dia para o mercado de jogos, torna-se possível explorar ainda mais as potencialidades associadas ao uso das superfícies de subdivisão. De facto, ao migrar os algoritmos de subdivisão de superfícies para os GPUs liberta-se o CPU desse fardo podendo o seu esforço ser inteiramente dedicado à simulação.

2. TRABALHO RELACIONADO

Durante muito tempo pensou-se que as superfícies obtidas por subdivisão recursiva não poderiam ser avaliadas analiticamente para valores arbitrários dos seus parâmetros. No entanto, Stam [Stam98] provou que tal é possível e usou como exemplo superfícies de Catmull-Clark. Apesar de se poder implementar a técnica num *fragment program*, o tratamento de todos os possíveis casos, relativamente a vértices extraordinários, fronteiras e cantos, resultaria num programa demasiado complexo [Zorin02].

Outros autores [Bolz02, Bolz03, Brickhill01] exploraram uma propriedade importante que permite a implementação de um algoritmo iterativo. Dado que a operação de *tesselation* de uma combinação linear de funções base é equivalente a uma combinação linear de *tesselations* de funções base, é possível, para qualquer nível de subdivisão, exprimir as posições dos vértices desse nível em função de combinações lineares dos pontos da malha inicial. Por outras palavras, o vector formado pelos vértices da superfície subdividida após vários passos de subdivisão recursiva pode escrever-se como o produto de potências da matriz de subdivisão pelo vector de pontos iniciais.

[Brickhill01] aplicou este princípio à subdivisão de malhas de triângulos usando a técnica de Loop, destinando-se a implementação à Playstation 2. Em [Bolz02] o mesmo princípio foi aplicado a quadriláteros, usando superfícies de Catmull-Clark. A primeira implementação corria no CPU, sendo posteriormente convertida para correr directamente no GPU, numa placa GeForce FX [Bolz03].

Comum às técnicas acima referidas, está a sua grande versatilidade, pois elas operam sobre malhas de topologia arbitrária, onde a valência dos vértices pode ser qualquer. No entanto, de modo a reduzir a complexidade da implementação é necessário um passo de pré-processamento para as malhas de topologia arbitrária. A ideia é a de isolar os vértices extraordinários (isto é, com valência diferente de 6 no caso da subdivisão de Loop ou diferente de 4 no caso das superfícies de Catmull-Clark) de tal forma que, para cada *patch* a subdividir, exista apenas um tal vértice não regular. Este processo requer a aplicação de 1 ou mais passos de subdivisão uniformemente em toda a malha (conforme o tipo de subdivisão em questão). Note-se que a aplicação de um passo de subdivisão apenas introduz vértices regulares permitindo assim o isolamento dos vértices irregulares. De seguida os *patches* que incorporam vértices irregulares são classificados de acordo com o caso específico.

Ora, é precisamente este pré-processamento da malha que impede a sua fácil utilização num contexto em que a malha original varia de forma dinâmica ao longo do tempo. No exemplo aqui em estudo, a utilização da variação dinâmica do nível de detalhe da malha usada na simulação de tecidos impede que estas técnicas se possam usar de forma simples e eficiente.

Por outro lado, a aplicação de um elevado número de passos de subdivisão de uma única vez pode ser indesejada, pois não permite *feedback* exterior nos passos intermédios. Tomando como exemplo a animação de tecidos, a subdivisão da malha usada na simulação pode criar vértices que introduzam intersecções com outros objectos ou até mesmo com ela própria. É mais simples tratar estas novas intersecções passo a passo (o seu número é mais reduzido) do que efectuar o procedimento para um número potencialmente bastante superior. Por exemplo, nas superfícies de Catmull-Clark, em cinco passos de subdivisão um quadrilátero dá origem a 256 quadriláteros na superfície final.

Assim, propomos neste artigo um algoritmo de subdivisão para ser usado no contexto da animação de superfícies deformáveis com variação do nível de detalhe. O nosso algoritmo, corre integralmente no GPU e aplica-se a malhas 4-8 regulares. Apesar de seguirmos uma implementação recursiva tradicional, de facto a nossa abordagem pode considerar-se mista, pois em cada passagem efectuamos dois passos de subdivisão.

O esquema de subdivisão do qual partimos é o da subdivisão 4-8 [Velho01]. No entanto as regras foram ligeiramente adaptadas por questões de eficiência e simplicidade que veremos adiante.

3. ESQUEMA DE SUBDIVISÃO

Nesta secção ilustraremos os passos que nos conduziram à obtenção do nosso esquema modificado de subdivisão de malhas 4-8.

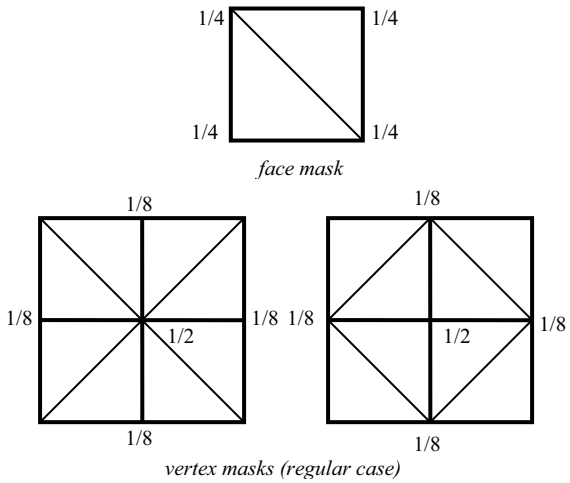


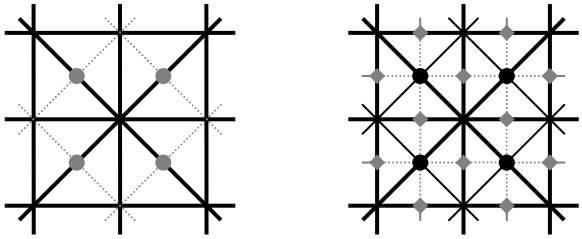
Figura 1 – Máscaras das regras de subdivisão 4-8 (caso regular)

As malhas 4-*k* [Velho00] são também conhecidas pelo termo *triquad meshes* e podem ser descritas por um agrupamento de triângulos, dois a dois, de modo a formarem um quadrilátero. Cada um destes quadriláteros define um bloco base e a operação de subdivisão elementar é a bissecção da sua aresta interior.

Fundamentalmente, os esquemas de subdivisão de superfícies decompõem-se em dois tipos de regras:

- *Face rules* – Adicionam novos vértices à superfície.
- *Vertex rules* – Filtram as posições dos vértices existentes.

Na Figura 1 podemos observar as máscaras associadas a cada uma das regras, acima mencionadas, para o caso de malhas 4-8 regulares (caso particular das malhas 4-*k*). As máscaras para as fronteiras de malhas fechadas serão analisadas mais adiante. A operação de bissecção corresponde à aplicação da *face rule* usando a primeira das máscaras da Figura 1 (*face mask*).Uma característica importante da subdivisão 4-8 é o facto da aplicação de dois passos de subdivisão corresponderem à quadrissecção dos quadriláteros que formam os blocos elementares, como se pode observar na Figura 2. Este facto é explorado na nossa implementação e conduz a programas muito simples, pois a disposição dos dados na memória do processador gráfico é muito natural. Refira-se, a propósito, que um dos grandes problemas na transposição de algoritmos para executarem nos GPUs é precisamente o da organização particular dos dados em memória. Na figura, os círculos cinzentos indicam os vértices introduzidos no respectivo passo da subdivisão.



1º passo *2º passo*

Figura 2 – Dois passos de subdivisão equivalem a uma quadrissecção dos blocos iniciais

Uma rápida análise da Figura 2 revela que, após os dois passos, podemos classificar os vértices resultantes da malha subdividida em três tipos:

- Tipo 1 - Vértices que substituem os vértices presentes na malha inicial. Estes vértices são o resultado da acumulação da dupla aplicação das *vertex rules*.
- Tipo 2 - Vértices que resultam da filtragem, no segundo passo, dos vértices criados durante o 1º passo (círculos pretos na Figura 2).
- Tipo 3 - Vértices introduzidos no segundo passo. Estes vértices estão assinalados a cinzento no lado direito da Figura 2.

Resta agora deduzir as máscaras para o esquema de subdivisão resultante da aplicação dos dois passos de seguida. O que necessitamos é de exprimir o cálculo das posições finais de cada tipo de vértice (dentro da classificação anterior) em função dos vértices da malha inicial.

A dedução das máscaras que permitem calcular cada um dos tipos de vértices é mostrada no Anexo A, apresentando-se aqui apenas o resultado final.

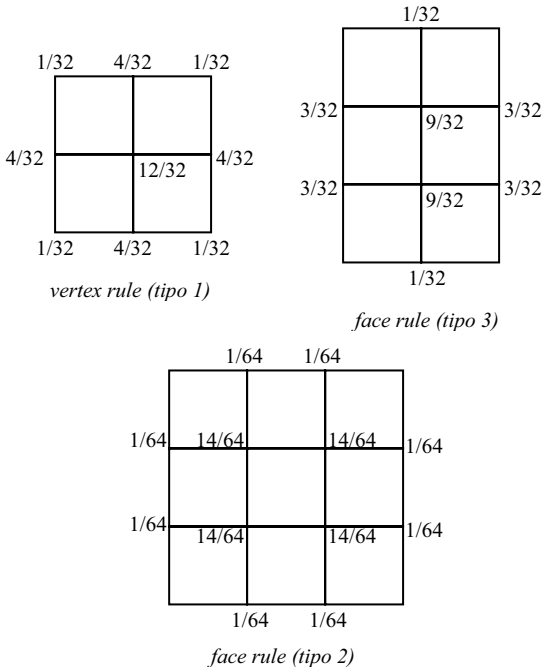


Figura 3 - Regras finais de subdivisão para o esquema de subdivisão 4-8 em dois passos

As regras são semelhantes, embora com pesos diferentes, às da subdivisão de Catmull-Clark para quadriláteros. No entanto, o suporte para algumas das regras é maior, no sentido de abranger mais pontos, o que se traduz numa desvantagem do ponto de vista da eficiência. Na secção seguinte mostraremos a nossa abordagem para contornar esta situação.

De modo a completar o esquema falta ainda deduzir as regras de subdivisão nos pontos fronteira e nos cantos. A dedução destas regras é efectuada tendo em conta o efeito final pretendido.

No contexto da simulação de tecidos ou outras superfícies deformáveis, o objectivo do uso da subdivisão de superfícies é simples: refinar, no final da simulação, a malha de controlo de forma a evitar a aparência angular que transparece quando surgem dobras ou rugas na superfície.

As regras de subdivisão a aplicar nas fronteiras da superfície terão que respeitar uma condição essencial, que é a de não provocarem o encolhimento do tecido a toda a sua volta.

Para além do mais, pode ser vantajoso refinar várias superfícies adjacentes (ou coladas nos bordos) de forma

independente sem que, na fronteira comum da superfície final subdividida, surjam buracos.

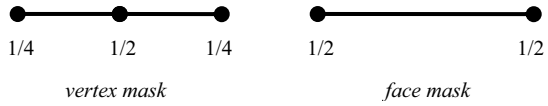


Figura 4 – Máscaras para as fronteiras

Com base nestas considerações, as nossas regras de subdivisão para as fronteiras resultam em combinações lineares onde apenas estão envolvidos os pontos da superfície original também situados na fronteira. Estas regras podem ser observadas na Figura 4.

No caso dos cantos, eles são inalterados pelas regras de subdivisão.

Na secção seguinte analisaremos a respectiva implementação no GPU juntamente com as modificações introduzidas no esquema de modo a obter programas mais eficientes e mais simples de implementar.

4. IMPLEMENTAÇÃO NO GPU

Os GPUs correntes possuem dois tipos distintos de processadores capazes de serem programados externamente:

- *Vertex processors* – permitem executar pequenos programas (*vertex programs* ou *vertex shaders*) que são activados por cada vértice das primitivas gráficas a desenhar.
- *Fragment processors* – permitem executar pequenos programas (*fragment programs* ou *pixel shaders*) que são activados por cada fragmento a escrever no framebuffer.

A grande vantagem dos processadores de fragmentos, no contexto da programação genérica nos GPUs, é o facto de poderem ser activados pelo CPU para operarem sobre grandes colecções de dados, gastando muito pouca largura de banda e tempo de CPU. A técnica usual consiste em desenhar um quadrilátero num framebuffer invisível (*offscreen*). Durante a fase de geração dos fragmentos (ou rasterização) será executado um programa, dentro dos processadores de fragmentos, por cada fragmento gerado.

Os vértices do quadrilátero poderão eventualmente activar a execução de programas nos processadores de vértices. O resultado de cada uma destas execuções é um vértice, provavelmente transformado. Numa situação em que se pretende processar um grande volume de dados não é de todo conveniente usar os processadores de vértices porque, por cada execução, é necessário fornecer um vértice do CPU para o GPU. Deste modo, o nosso algoritmo corre, naturalmente, nos processadores de fragmentos.

Um dos grandes problemas quando se pretende programar um algoritmo no GPU reside nas grandes limitações do paradigma de *stream computing*. O mesmo programa é executado para múltiplos registos de activação (um *stream*). A execução é efectuada em paralelo nos diferentes processadores disponibilizados pela placa sem qualquer comunicação entre eles. Não existem registos ou qualquer outro tipo de memória global partilhada para escrita. No caso dos programas de fragmentos, o output de cada programa é um pixel no framebuffer de saída. Os diferentes processadores vão operando em paralelo gerando pixels distintos. Também não é possível ao programa decidir qual o pixel a escrever no final, pois tal é imposto pela rasterização da primitiva gráfica invocada.

Para além de um conjunto de instruções para operações numéricas em números de vírgula flutuante, existe um outro conjunto de operações dedicadas ao processamento de texturas. Quando pensamos em programação genérica as texturas servem, essencialmente, para guardar as suas estruturas de dados. Ora, a organização dos dados em texturas é, em muitos casos, um dos grandes obstáculos à programação dos GPUs.

O interesse crescente na programação de GPUs para tarefas normalmente atribuídas ao CPU faz desta área um turbilhão de novos desenvolvimentos, quer em termos de hardware, quer em termos de software. As capacidades dos processadores de vértices e de fragmentos está em permanente evolução. A cada geração de novas placas mais e melhor funcionalidade é incorporada nestes processadores. Consequentemente, as APIs que permitem programar os GPUs estão também em constante evolução e não é de estranhar a panóplia de extensões ao OpenGL, que permitem tirar partido do hardware mais recente, ou das constantes actualizações do Direct3D.

A nossa implementação foi realizada sobre OpenGL tendo os programas (*shaders*) sido escritos usando a linguagem Cg da nVIDIA. Algumas das extensões mais importantes do OpenGL usadas na nossa implementação são:

- EXT_pbuffer – permite a criação de frame buffers invisíveis para rendering *offscreen*.
- NV_float_buffer – permite a criação de frame buffers (*pbuffers*) e texturas formadas por valores em vírgula flutuante sem restrições de limites aos seus valores que não os da própria representação binária.
- ARB_render_texture – permite que o conteúdo de um *pbuffer* possa ser usado directamente como uma textura. Oferece assim um mecanismo de realimentação onde os dados escritos em passos anteriores possam ser processados por novos programas de fragmenteos dentro do GPU sem haver necessidade de uma viagem de ida e volta até ao CPU.

4.1. Organização dos dados em texturas

No caso do nosso algoritmo, como ele funciona sobre malhas 4-8 regulares e estas têm uma estrutura exterior

formada por quadriláteros, o mapeamento das nossas malhas de vértices para a memória do GPU poderia ser a mais natural possível. A malha seria representada sob a forma de textura 2D armazenada no formato de vírgula flutuante. Cada *texel* representaria um vértice, encontrando-se os vértices vizinhos dos quadriláteros da malha nas posições vizinhas do respectivo *texel*.

No entanto, como se verá adiante, esta primeira disposição dos dados faz-nos incorrer numa grande penalização em termos de eficiência, obrigando-nos a pensar numa estratégia diferente para guardar as malhas 4-8 regulares.

A eficiência dos programas de fragmentos está directamente ligada ao número de instruções que os respectivos processadores têm que executar. Os acessos à memória (texturas) exibem uma latência elevada, mas o facto está escondido graças a um *pipeline* muito extenso. Deste modo, a preocupação fundamental está na redução do número de instruções que os programas terão que executar.

O programa mais simples funcionaria desenhando um *quad* de dimensões $(2w\text{-}1) \times (2h\text{-}1)$ pixels, para uma malha regular inicial de tamanho $w \times h$ guardada numa textura rectangular 2D. Dentro do programa far-se-ia a identificação do tipo de pixel (vértice) a gerar e utilizar-se-iam as máscaras apropriadas (ver Figura 3 e Figura 4).

O problema desta estratégia, é que o suporte para *branching* nos processadores de fragmentos é inexistente[1]. Os saltos dinâmicos são simulados com escritas condicionais. As diversas execuções possíveis do programa são expandidas e todos os ramos são executados. No final, apenas se aproveitam para escrita no output os valores do ramo que seria efectivamente seguido. Se considerarmos que o número de instruções para calcular cada o tipo de vértice da malha refinada usa aproximadamente o mesmo número de instruções, a execução do *shader* seria cerca de 4 vezes mais lenta devido à inexistência de *branching*. O valor 4 obtém-se deixando de fora as regras para as fronteiras (caso contrário seria ainda pior) e considerando os 3 tipos de regras da Figura 3. Note-se que a regra do tipo 3 na realidade é um par de regras. Uma para os vértices gerados ao longo das arestas horizontais da malha e outra igual mas rodada 90º para os vértices gerados ao longo das arestas verticais.

Deste modo, propomos um *layout* das malhas regulares dentro de texturas diferente do acima mencionado. A ideia é usar um shader diferente para cada tipo de vértice a gerar, evitando-se, assim, o *branching*. No caso dos vértices interiores da malha, existem quatro tipos distintos, dando origem a outros tantos programas de fragmentos. Esta estratégia obriga-nos a concentrar no *pbuffer* de saída todos os vértices do mesmo tipo.

[1] Na realidade, as placas da 5ª geração tais como as GeForce 6000 já possuem alguma capacidade, embora limitada, para efectuar saltos dinâmicos nos *fragment programs*.

Note-se que a escrita no pixel buffer em coordenadas aleatórias é impossível. Cada programa escreve apenas numa localização pré-determinada pelo processo de varrimento da primitiva gráfica.

Considere-se uma malha regular de $w \times h$ vértices. O resultado final de um passo de subdivisão do nosso esquema modificado será disposto numa textura, de dimensão $(2w\text{-}1) \times (2h\text{-}1)$, exibindo a estrutura de blocos ilustrada na Figura 5.

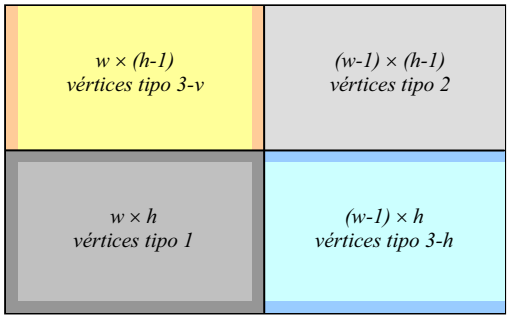


Figura 5 - Layout de uma malha regular sob a forma de textura

Os bordos mais escuros de cada zona correspondem a vértices que se situam na fronteira, pelo que é necessário usar as variantes apropriadas dos respectivos programas principais.

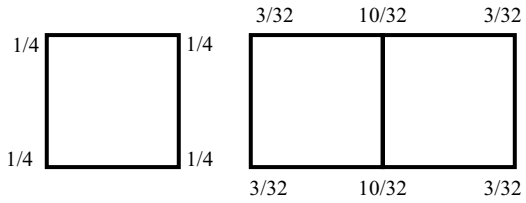
Obviamente, a malha de entrada de dimensão $w \times h$ é, também ela, representada da mesma forma dentro de uma textura caso contrário seria necessário, no final de cada passo, voltar a mudar a representação da malha na memória do GPU.

4.2. *Fragment Programs*

Não há qualquer dificuldade acrescida pelo facto de as malhas se encontrarem organizadas desta forma na memória. Do ponto de vista dos programas de fragmentos para cada uma das regras, eles serão invocados separadamente, desenhando um quadrilátero que cubra cada um dos blocos acima ilustrados (claro que as fronteiras também terão os seus próprios programas específicos).

Na tentativa de reduzir ainda mais a complexidade dos nossos programas de fragmentos procedemos ainda a uma simplificação das nossas regras. Na secção 3 mostrámos as máscaras associadas a cada regra de subdivisão. Estas máscaras exibem um suporte superior às do esquema de subdivisão de Catmull-Clark. O problema do suporte alargado é que o número de pontos envolvidos no cálculo é superior e isso traduz-se num maior número de acessos a texturas e em mais operações aritméticas. A juntar a isto, as regras obtidas na Secção 3 (Figura 3) introduzem uma complicação adicional ao criarem dois casos especiais para as fronteiras. De facto, a implementação directa das regras conduziria a programas distintos para as duas filas de vértices mais exteriores.

Decidimos por isso simplificar as regras do tipo 2 e do tipo 3 que são as que fazem explodir a complexidade do número de casos na fronteira da malha. Se repararmos, os vértices mais exteriores dessas máscaras têm um peso muito pequeno e no nosso esquema modificado resolvemos transferir esse peso para os vértices vizinhos mais interiores, mantendo a simetria das máscaras. O resultado final pode ser observado na Figura 6. Não observámos quaisquer desvantagens da utilização das regras modificadas em termos das propriedades que pretendíamos para as superfícies geradas.



face rule (tipo 2) *face rule (tipo 3)*

Figura 6 - Regras finais (após simplificação) do nosso esquema de subdivisão 4-8 modificado

A título meramente ilustrativo, dada a simplicidade dos programas obtidos, apresenta-se de seguida o programa que efectua os cálculos dos vértices tipo 2. Os outros casos são semelhantes, apenas envolvendo mais alguns pontos.

```
float4 main(float2 uv: TEXCOORD0,
            uniform samplerRECT mesh,
            uniform float2 dims): COLOR
{
     float4 p1 = texRECT(mesh, mapToQuadType1(uv, dims));
     float4 p2 = texRECT(mesh, mapToQuadType2(uv, dims));
     float4 p3 = texRECT(mesh, mapToQuadType3h(uv, dims));
     float4 p4 = texRECT(mesh, mapToQuadType3v(uv, dims));
     return (p1+p2+p3+p4)/4;
}
```

Nos nossos programas optámos por especificar as coordenadas de textura dos vértices dos quads de tal forma que elas coincidam com as coordenadas inteiras dos pixels (de modo a que a origem seja o canto inferior esquerdo do quad). As funções auxiliares mapToQuadTypeX são muito pequenas e simples e, para cada pixel a gerar no pbuffer, indexam um ponto de referência apropriado em cada um dos blocos. O vector 2D *dims* contém as dimensões da malha do passo anterior.

5. DETALHES DO ALGORITMO DO CPU

Nesta secção analisaremos a parte do algoritmo assistida pelo CPU. Na realidade, para além da detecção da condição de terminação e das ordens para gerar os sucessivos *quads*, pouco mais há a fazer do lado do CPU.

Considerando uma malha regular inicial de dimensões $R_w \times R_h$ e, uma resolução final pretendida, não superior a $M_w \times M_h$, o algoritmo começa por criar dois pbuffers com as dimensões máximas $M_w \times M_h$. Na realidade um dos pbuffers pode ser reduzido para um quarto desde que tenhamos o cuidado de colocar a malha inicial no buffer certo de modo a que no final o buffer que contém o resultado seja o maior. Note-se que a cada passo do algoritmo se dá a troca dos papéis dos dois pbuffers.

```
// Activar os processadores de fragmentos para
// ambos os pbuffers

// As dimensões correntes da malha são as dimensões da
// malha regular de entrada
Fw = Rw; Fh = Rh

// Enquanto o pbuffer for suficientemente grande para
// mais um passo de subdivisão…
while((Fw < Mw/2) && (Fh < Mh/2)) {

     // Cálculo das dimensões da nova malha
     Fw = 2*Fw-1;
     Fh = 2*Fh-1;

     // Tornar o buffer de saída o framebuffer corrente
     ob->Activate();

     // usar o buffer de input como textura
     wglBindTexImageARB(ib->getHPBufferARB(),
                        WGL_FRONT_LEFT_ARB);

     // Chamar a função que desenha os quatro quads
     // usando os programas referidos na secção 4.2
     Subdivide2Steps(Fw, Fh, ib, ob);

     // libertar o buffer de input da textura
     wglReleaseTexImageARB(ib->getHPBufferARB(),
                           WGL_FRONT_LEFT_ARB);

     // Desactivar o buffer de output
     ob->Deactivate();

     // Trocar os papéis dos buffers
     // de input (ib) e de output (ob)
     Swap(ib, ob);

     steps++;
}
```

De notar que no nosso algoritmo não é obrigatório efectuar a subdivisão da malha de uma só vez para o nível de subdivisão final pretendido. Na realidade, pode-se efectuar a subdivisão por blocos, caso o nível de subdivisão pretendido seja tal que o GPU não possa suportar todas as texturas carregadas em simultâneo na sua memória. Mais ainda, devido à sua natureza recursiva, é sempre possível pegar em qualquer malha obtida por subdivisão e voltar a subdividi-la.

6. MALHAS COM LOD DINÂMICO

No caso do nosso simulador de tecidos, o modelo dinâmico utilizado tem por base o proposto por Baraff e Witkin [Baraff98]. É usada a mesma técnica de integração numérica que permite efectuar grandes passos na simulação de cada vez.

O nível de detalhe das nossas malhas varia dinamicamente ao longo da simulação [Birra04] e para tal tiramos partido da utilização de malhas 4-*k* que são uma generalização das malhas 4-8 regulares. Zonas da malha onde são detectados grandes efeitos angulosos são subdivididas pelo CPU, localmente. A ideia é usar um maior número de partículas onde de facto elas parecem ser mais necessárias para capturar as rugas e dobras dos tecidos.

De modo a podermos usar o algoritmo de pós-processamento das malhas 4-8 aqui proposto torna-se necessário subdividir a malha da simulação para que os triângulos pertençam todos ao mesmo nível de detalhe.

Uma solução barata, totalmente oferecida pelo hardware de interpolação do GPU, passa por efectuar o rendering da malha usando as coordenadas 3D dos respectivos vértices como se de coordenadas de textura se tratassem.

Como posição para os vértices usam-se as coordenadas paramétricas da malha.

Apesar de simples, a abordagem revelou ter aplicação prática quando em presença de uma gama relativamente pequena de níveis de detalhe. Se a amplitude de níveis de detalhe for muito grande acabamos por conseguir notar o efeito anguloso da superfície pois o processo de uniformizar o nível de detalhe vai gerar muitos pontos co-planares para cada zona inicialmente plana da malha original. O efeito é que a superfície é suavizada mas a curvatura tem um raio muito pequeno porque os pontos que provocam a aparência angulosa estão muito próximos espacialmente.

É importante realçar que este processo ocorre totalmente dentro do GPU. A malha resultante da uniformização está num pbuffer da placa gráfica, sendo este usado como textura no primeiro passo recursivo do nosso algoritmo de subdivisão.[2]

7. RESULTADOS

Nas Figuras 7, 8 e 9 podemos observar o resultado prático da aplicação desta técnica. Uma zona angulosa do tecido foi ampliada de modo a que se pudessem observar individualmente os pontos gerados pelo nosso algoritmo de subdivisão. Note-se a ausência de artefactos e a elevada suavidade da malha final. Na Figura 8 podemos observar com grande clareza o efeito suavização da malha original. Pequenas covas ou bicos são tapados ou cortados pela malha subdividida.

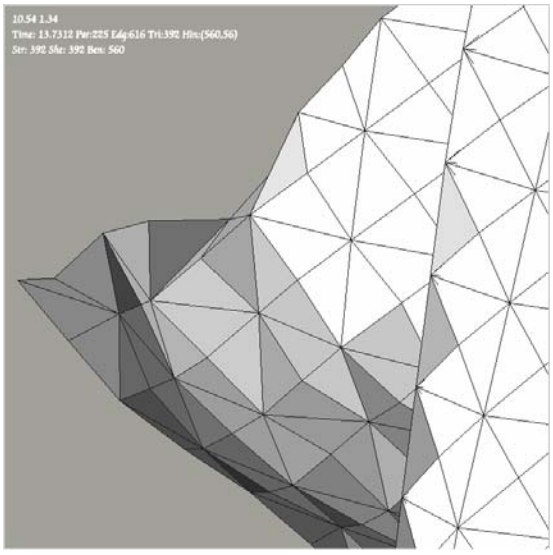


Figura 7 – malha polygonal 4-8 regular usada numa simulação

Em termos de performance medimos o débito de vértices dos nossos *shaders* no último passo de subdivisão que leva à criação de uma malha de 497x497 vértices. O mesmo *shader* foi executado 1000 vezes sem que houvesse mudanças de framebuffer ou outras tarefas pesadas de modo a medir com maior precisão a *performance*. No caso de se usar um único programa ingénuo para processar todos os 4 tipos de vértices, o tempo obtido para o teste atrás referido foi de 80s. Usando os programas separados para cada tipo de vértice o tempo baixa para pouco mais de 1/4 (27s). Todos os testes foram realizados numa placa GeForce FX Go5200 (a menos rápida da penúltima geração de placas da nVIDIA). Estes valores representam um débito sustentado de aproximadamente 9 milhões de vértices por segundo.

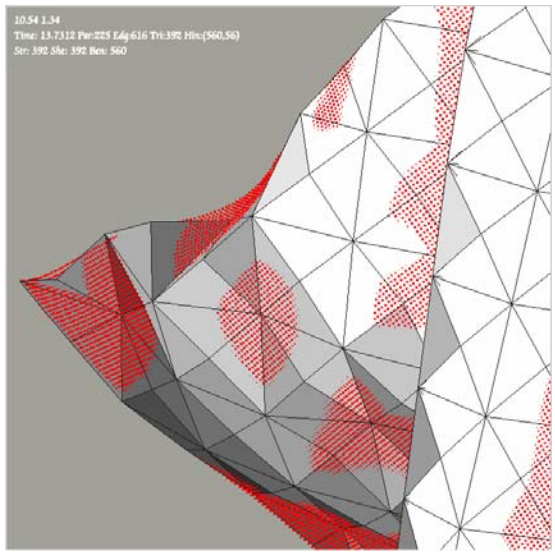


Figura 8 – A mesma malha da figura anterior mas agora com os pontos resultantes da subdivisão

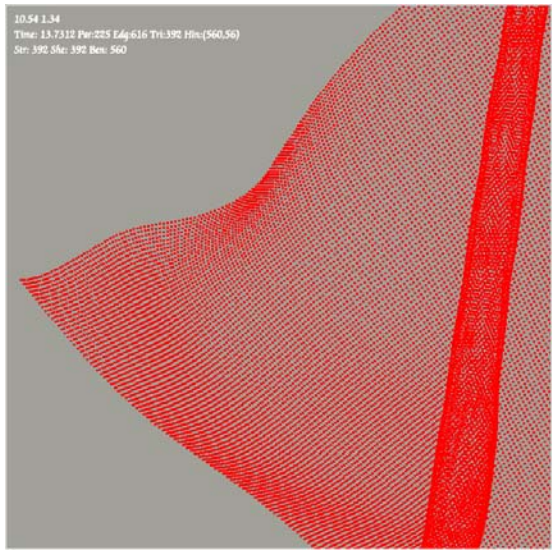


Figura 9 – Apenas os pontos gerados pela subdivisão para a malha da Figura 7

O mesmo algoritmo, com algumas optimizações de eficiência, demora 70s, para efectuar o mesmo teste, num Pentium 4 com Hyperthreading a 3.0Ghz e bus de memória a 800Mhz. Note-se que a placa gráfica usada é muito limitada, quer em termos de unidades de processamento paralelas quer em termos de memória

[2] Na realidade é necessário invocar um shader que disponha os dados de acordo com o layout definido na subsecção 4.1.

(tamanho e velocidade de acesso), sendo claramente batida por modelos mais recentes.

Na Figura 10, podemos observar uma malha, não regular, com variação dinâmica do nível de detalhe. Após o processo de uniformização da malha, conforme descrito na secção 6, procedeu-se à sua subdivisão, sendo o resultado ilustrado na Figura 11.

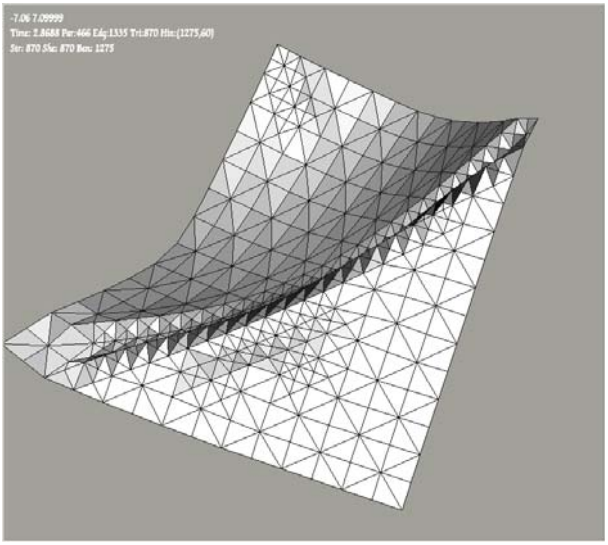


Figura 10 – Malha de simulação com variação dinâmica de LOD

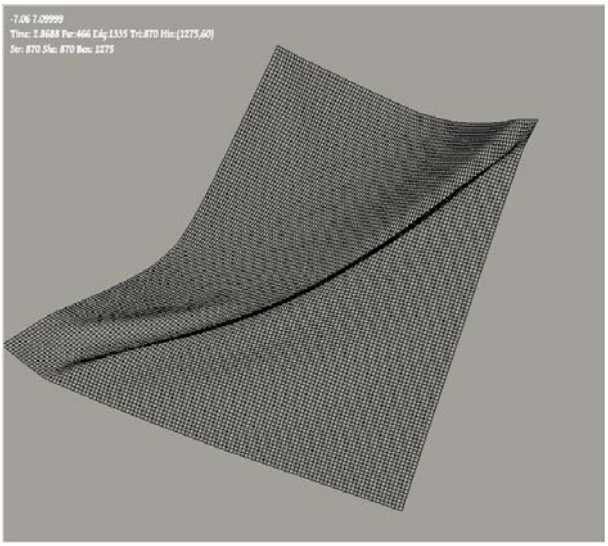


Figura 11 – A malha da figura anterior depois de subdividida

8. CONCLUSÕES E TRABALHO FUTURO

Neste artigo apresentámos um novo algoritmo recursivo de subdivisão de malhas 4-8 regulares. As características principais do algoritmo são resultantes da fusão de dois passos de subdivisão de cada vez. De modo a optimizar a eficiência do método deduzido procedemos à simplificação de algumas das regras de subdivisão sem aparente perda das características fundamentais pretendidas, principalmente a grande suavidade da malha final obtida.

Apesar de o algoritmo ser recursivo, resultando numa perda de desempenho quando comparado com o método proposto em [Bolz03], tem a vantagem de não ser necessário carregar previamente o GPU com texturas gigantescas contemplando os diferentes casos de topologia. Além do mais o algoritmo apenas requer uma mudança de contexto do OpenGL (operação pesada aliada à activação dos pbuffers) entre cada aplicação recursiva do mesmo, podendo cada passo subdividir um grande número de patches. No caso do algoritmo proposto em [Bolz03] é necessária uma mudança de contexto por cada patch, sendo esses tempos descontados pelos autores na análise da performance do seu algoritmo. A justificação, válida porém, é que no futuro os drivers optimizarão o processo. Outras extensões do OpenGL, emergentes, poderão tornar o processo de comutação de pbuffers numa operação mais leve, tais como a novíssima extensão EXT_framebuffer (apenas disponível em drivers beta).

Como trabalho futuro, propomos a investigação do problema da detecção de colisões nos passos de subdivisão sucessiva. Para tirar partido do algoritmo aqui apresentado é necessário que esta operação de validação do estado da malha - detectar que não existem novas colisões introduzidas pelo passo de subdivisão – se processe totalmente dentro do GPU. Existe já algum trabalho nesse sentido embora ainda esteja numa fase muito embrionária. O problema é de fácil solução se ignorarmos as auto-colisões e apenas existir um número reduzido de objectos na cena.

ANEXO A

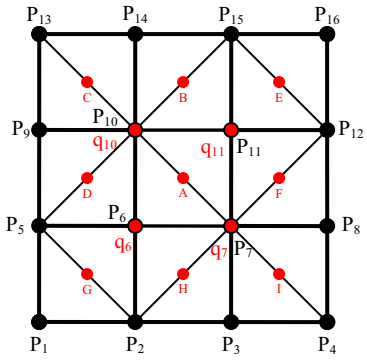


Figura 12 – Primeiro passo de subdivisão

Vértices criados durante o primeiro passo:

A = 1/4 ($\mathbf{P}_6 + \mathbf{P}_7 + \mathbf{P}_{10} + \mathbf{P}_{11}$)

B = 1/4 ($\mathbf{P}_{14} + \mathbf{P}_{15} + \mathbf{P}_{10} + \mathbf{P}_{11}$)

C = 1/4 ($\mathbf{P}_{13} + \mathbf{P}_{14} + \mathbf{P}_9 + \mathbf{P}_{10}$)

D = 1/4 ($\mathbf{P}_9 + \mathbf{P}_{10} + \mathbf{P}_5 + \mathbf{P}_6$)

E = 1/4 ($\mathbf{P}_{15} + \mathbf{P}_{16} + \mathbf{P}_{11} + \mathbf{P}_{12}$)

F = 1/4 ($\mathbf{P}_{11} + \mathbf{P}_{12} + \mathbf{P}_7 + \mathbf{P}_8$)

G = 1/4 ($\mathbf{P}_5 + \mathbf{P}_6 + \mathbf{P}_1 + \mathbf{P}_2$)

H = 1/4 ($\mathbf{P}_6 + \mathbf{P}_7 + \mathbf{P}_2 + \mathbf{P}_3$)

I = 1/4 ($\mathbf{P}_7 + \mathbf{P}_8 + \mathbf{P}_3 + \mathbf{P}_4$)

Vértices filtrados durante o primeiro passo:

$\mathbf{q}_{10} = 1/2\ \mathbf{P}_{10} + 1/8\ (\mathbf{P}_{14}+\mathbf{P}_6+\mathbf{P}_9+\mathbf{P}_{11})$

$\mathbf{q}_{11} = 1/2\ \mathbf{P}_{11} + 1/8\ (\mathbf{P}_{15}+\mathbf{P}_7+\mathbf{P}_{10}+\mathbf{P}_{12})$

$\mathbf{q}_6 = 1/2\ \mathbf{P}_6 + 1/8\ (\mathbf{P}_{10}+\mathbf{P}_2+\mathbf{P}_5+\mathbf{P}_7)$

$\mathbf{q}_7 = 1/2\ \mathbf{P}_7 + 1/8\ (\mathbf{P}_{11}+\mathbf{P}_3+\mathbf{P}_6+\mathbf{P}_8)$

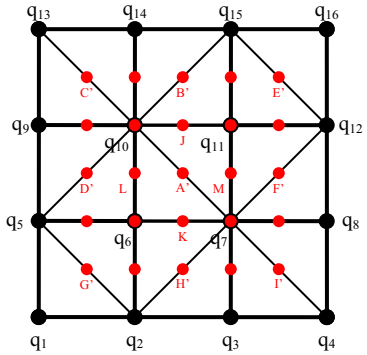


Figura 13 – Segundo passo de subdivisão

Vértices criados durante o segundo passo:

$\mathbf{J} = 1/4\ (\mathbf{A} + \mathbf{B} + \mathbf{q}_{10} + \mathbf{q}_{11})$

$\mathbf{K} = 1/4\ (\mathbf{A} + \mathbf{H} + \mathbf{q}_6 + \mathbf{q}_7)$

$\mathbf{L} = 1/4\ (\mathbf{A} + \mathbf{D} + \mathbf{q}_{10} + \mathbf{q}_6)$

$\mathbf{M} = 1/4\ (\mathbf{A} + \mathbf{F} + \mathbf{q}_{11} + \mathbf{q}_7)$

Vértices filtrados durante o segundo passo:

$\mathbf{r}_6 = 1/2\ \mathbf{q}_6 + 1/8\ (\mathbf{D} + \mathbf{H} + \mathbf{A} + \mathbf{G})$

$\mathbf{r}_7 = 1/2\ \mathbf{q}_7 + 1/8\ (\mathbf{A} + \mathbf{I} + \mathbf{H} + \mathbf{F})$

$\mathbf{r}_{10} = 1/2\ \mathbf{q}_{10} + 1/8\ (\mathbf{A} + \mathbf{B} + \mathbf{C} + \mathbf{D})$

$\mathbf{r}_{11} = 1/2\ \mathbf{q}_{11} + 1/8\ (\mathbf{A} + \mathbf{B} + \mathbf{E} + \mathbf{F})$

$\mathbf{A'} = 1/2\ \mathbf{A} + 1/8\ (\mathbf{q}_{10} + \mathbf{q}_7 + \mathbf{q}_6 + \mathbf{q}_{11})$

Expandindo as expressões para cada tipo de vértice:

$\mathbf{r}_6 = 1/32\ (\mathbf{P}_1 + 4\mathbf{P}_2 + \mathbf{P}_3 + 4\mathbf{P}_5 + 12\mathbf{P}_6 + 4\mathbf{P}_7 + \mathbf{P}_9)$

$\mathbf{K} = 1/32\ (3\mathbf{P}_2 + 3\mathbf{P}_3 + \mathbf{P}_5 + 9\mathbf{P}_6 + 9\mathbf{P}_7 + \mathbf{P}_8 + 3\mathbf{P}_{10} + 3\mathbf{P}_{11})$

$\mathbf{L} = 1/32\ (\mathbf{P}_2 + 3\mathbf{P}_5 + 9\mathbf{P}_6 + 3\mathbf{P}_7 + 3\mathbf{P}_9 + 9\mathbf{P}_{10} + 3\mathbf{P}_{11} + \mathbf{P}_{14})$

$\mathbf{A'} = 1/64\ (\mathbf{P}_2 + \mathbf{P}_3 + \mathbf{P}_5 + 14\mathbf{P}_6 + 14\mathbf{P}_7 + \mathbf{P}_8 + \mathbf{P}_9 + 14\mathbf{P}_{10} + 14\mathbf{P}_{11} + \mathbf{P}_{12} + \mathbf{P}_{14} + \mathbf{P}_{15})$

REFERÊNCIAS

[Baraff98] David Baraff and Andrew Witkin. Large Steps in Cloth Simulation. In *SIGGRAPH 98 Conference Proceedings*, Annual Conference Series, pages 43-54. ACM SIGGRAPH, 1998.

[Birra04] Fernando P. Birra, M. Próspero dos Santos. Dynamic Mesh Resolution and Stable Cloth Simulation. *In Advances in Computer Graphics in Portugal, Special Edition of the VIRTUAL Journal, CD-ROM*, November/December, 2004, ISBN 972-98464-5-6, ISSN 0873-1837. Online version: http://virtual.inesc.pt/aicg04

[Bolz02] J.Bolz, and P.Schröder. Rapid Evaluation of Catmull-Clark Subdivision Surfaces. *In Proceedings of Web3D*, 11-17. 2002.

[Bolz03] J.Bolz, and P.Schröder. Evaluation of Subdivision Surfaces on Programmable Graphics Hardware, 2003. http://multires.caltech.edu/pubs/GPUSubD.pdf

[Brickhill01] Pratical Implementation Techniques for Multi-Resolution Subdivision Surfaces. *In Game Developers Conference*.

[Bridson02] R. Bridson, R. Fedkiw, and J. Anderson. Robust Treatment of Collisions, Contact and Friction for Cloth Animation. *ACM Transactions on Graphics*, 21(3): 594–603, 2002.

[Catmull78] E. Catmull, and J. Clark. Recursively Generated B-Spline Surfaces on Arbitrary Topological Meshes. *Computer Aided Design 10*(6): 350-355. 1978.

[Doo78] D. Doo, and M.A.Sabin. Behaviour of Recursive Subdivision Surfaces Near Extraordinary Points. *Computer Aided Design 10*(6):356-360. 1978.

[Loop87] C.T. Loop. *Smooth Subdivision Surfaces Based on Triangles*. Master's Thesis, Department of Mathematics, University of Utah, August 1987.

[Loop01] C.Loop. Triangle Mesh Subdivision with Bounded Curvature and the Convex-Hull Property. *Tech.Rep.MSR-TR-2001-24.* Microsoft Research. 2001.

[Stam98] Jos Stam. Exact Evaluation of Catmull-Clark Subdivision Surfaces at Arbitrary Parameter Values. In *SIGGRAPH 98 Conference Proceedings*, Annual Conference Series, pages 395-404. ACM SIGGRAPH, 1998.

[Velho00] Luis Velho and Jonas Gomes. Variable Resolution 4–k Meshes: Concepts and Applications. *Computer Graphics Forum*, 19(4):195-214, 2000.

[Velho01] Luis Velho and Denis Zorin. 4–8 Subdivision. *Computer Aided Geometric Design*, 18(5): 397-427, 2001. Special Issue on Subdivision Techniques.

[Zorin02] D.Zorin, and D.Kristjansson. Evaluation of Piecewise Smooth Subdivision Surfaces. *The Visual Computer 18*, 5-6, 299-315. 2002.